Methodology: The Experts Speak

Five prominent software engineers discuss the methodologies for which they are famous

Ken Orr, Chris Gane, Edward Yourdon, Peter P. Chen, and Larry L. Constantine

The Warnier/Orr Approach

Ken Orr

It's more or less impossible to write about the Warnier/Orr methodology because, in fact, there is no such thing. While there are Warnier/Orr diagrams, Warnier's methodology (i.e., logical data structure, logical construction of systems, and logical construction of programs), and Orr's methodology (data-structured systems development), there is not, strictly speaking, a Warnier/Orr methodology.

Many software engineers confuse diagrams with methodologies. Perhaps this is natural, since the diagrams are the most visible part of most methodologies; but it's unfortunate, for methodologies are much more than just a set of diagrams and syntax rules.

Within the context of software engineering, a *method* is a procedure or technique for performing some significant portion of the software life cycle. Over the years, techniques have been developed for requirements definition, database design, program design, test-case development, and so on. A *methodology*, in software engineering terms, is a collection of methods based on a common philosophy that fit together in a framework called the systems development *life cycle*.

Methods often use a variety of tools: diagrams, forms, and text for documenting and communicating. Not surprisingly, these diagrams and forms often take on a life of their own. Diagrams, like words, can be used out of context, without understanding the purpose for which they were intended. While the results can be confusing, new possibilities and uses often arise that are quite fortuitous.

People who develop software engineering methods and methodologies attempt to solve problems, observe what others do, and derive, or abstract, patterns from all this. Those patterns ultimately turn into methodologies.

In my experience, my colleagues and I always know *what* works long before we know *why* it works. Software engineering methodologists are skilled at working with experts, such as analysts, programmers, database administrators, and so forth, finding out how these experts do what they do, and putting these findings down in such a way that others can follow them.

The correct name for what many people call the Warnier/Orr methodology is data-structured systems development. DSSD, like most methodologies, is actually the result of many people's efforts, in addition to my own, including my coworkers at Optima, colleagues, and clients. Much of the methodology has come about by taking various component technologies, such as structured programming and relational-database design, and putting them together into a coherent framework.

A Little History

In 1972, Terry Baker's article "Chief Programmer Team Operations" in the *IBM Systems Journal* had a major impact on the field. It brought together several ideas: structured programming, top-down design and implementation, the chief programmer, the chief-programmer team, and the documentation librarian. If there was a shot that started the "structured revolution" in the U.S., Baker's article was it.

In the early 1970s, I became interested in structured programming and in structured design. In applying the principles of top-down design, I discovered that many of my best, most intelligible solutions were those in which the hierarchi-

continued

cal structure of the program mirrored the hierarchical structure of the data the program was processing.

Shortly after this discovery, I stumbled across the work of Jean Warnier and realized that he not only had made the same discovery with regard to data-structured programming but had already built a systematic methodology around it. I also followed Michael Jackson's work, another form of data-structured design.

I already believed that you could and should construct programs hierarchically using only a few basic logical structures. Moreover, I believed that if you were going to build very large things, you should build them in systematic ways based on simple structures. This coincided with design and construction techniques used in fields such as electrical engineering. Structured programming represented a base on which to build; therefore, using the data structure as the framework for building the program structure seemed like the next natural step.

Data-structured programming meant that you could create predictably correct solutions r a wide class of programming problems—problems in which the structures of the input and the output were the same or very similar. But beyond that, Warnier, Jackson, and those of us involved in developing DSSD were able to extend data-structured techniques to arbitrarily complex programs.

To solve these more complex problems, you must recognize that the nature of the problem of complexity is, on one level at least, fundamentally mathematical in nature—that is, complex problems are fundamentally *n:n* (many-to-many) mappings from input to output. To deal with this complexity systematically, you must break the problems down into a series of less complex mappings.

This is what mathematicians have been doing for thousands of years—breaking large troublesome problems into smaller ones for which there are clear precise answers. In the case of data-structured design, this meant developing a scheme in which the physical inputs were mapped into logical inputs; the logical inputs were then mapped into the logical outputs; and, finally, the logical outputs were mapped into the physical outputs.

With this overall program-design framework comes a goal-oriented design strategy—an approach that starts with the structure of the output and works backward, first to the logical, or ideal, input, and then to the physical input.

The data-structured approach to program design has proven to be successful on a wide variety of problems, but it is clearly no panacea. What it does represent is a systematic approach to attacking complex problems (simple problems have a way of taking care of themselves, or, alternatively, becoming complex).

Programming in the Large

At some point in developing techniques for building systems, you realize that the most significant problems in software occur not at the programming level but at the systems level. How do you design entire suites of programs so that they work effectively together? How do you get the right requirements? Where does planning fit into the scheme of things?

Little by little, DSSD moved from a program-design methodology to a systems-design methodology. Over a period of years, the methodology was expanded to deal with database design, require-

continued

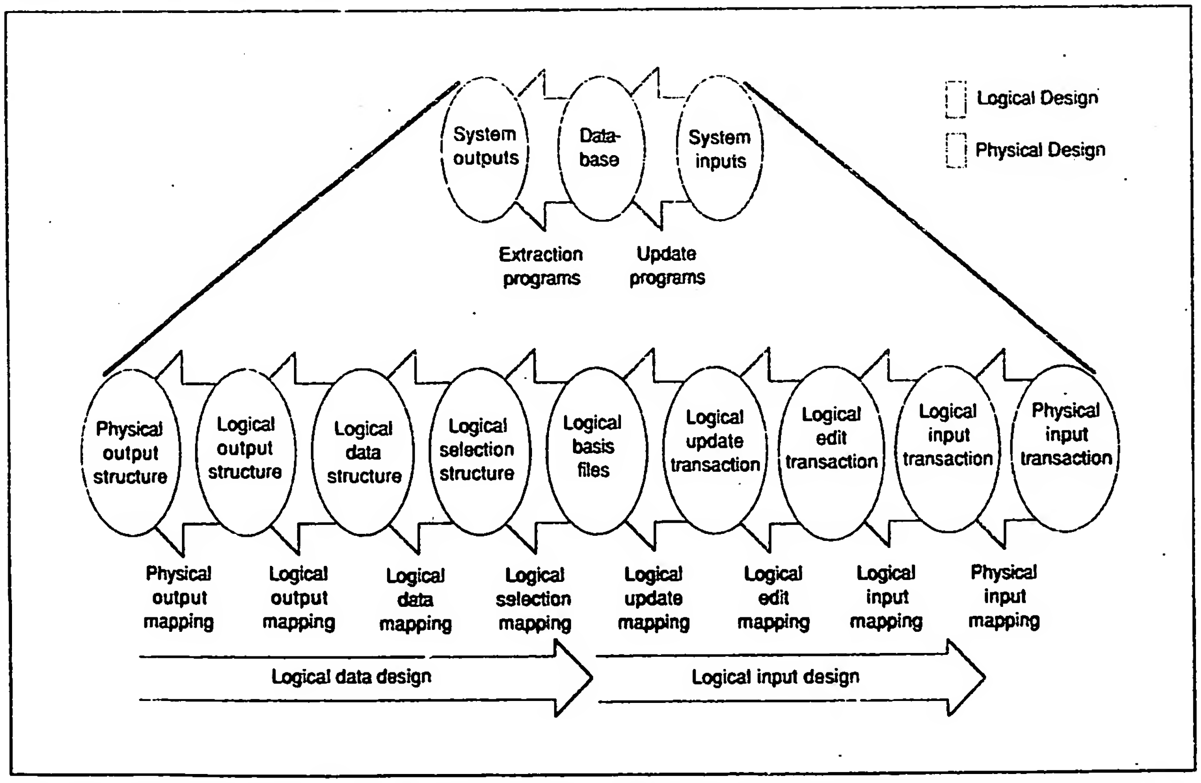


Figure 1: *At the systems level, instead of working backward to the ideal inputs, DSSD works backward to the logical database and then to the inputs.*

ay The Air Difference

MiT PC/XT SYSTEM

10MHz, 640K Memory, 150 Watt Power Supply, AT Style Keyboard, 1-360K Drive, AT-Style Case, Monographics Card with Parallel Printer Port, Clock Card with Serial Port, Mono Monitor with Swivel Tilt Base

$709

System w/2-360K Drives $787

System w/1-360K DR 1-20 Meg $994
System w/1-360K DR 1-30 Meg ... $1021

IBM SOFTWARE

WORD PROCESSORS

Easy Extra	$53
Grandview	175
Multimate Advantage II	275
PFS: Professional Write	102
Volkswriter 4	69
Volkswriter Deluxe Plus	57
Word (Microsoft)	179
Word Perfect 4.2	198
Word Perfect 5.0	219
Word Perfect Executive	117
Word Perfect Library	53
Wordstar Pro Pack	233
Wordstar 2000 Plus 3.0	239

DISKETTES

Maxell	Call
Sony 5¼ Qty. 100 DS/DD	$55
Sony 3½ Qty. 100 DD	145
Sony 3½ HD Qty. 100	355

DESKTOP PUBLISHING

Pagemaker 3.0	$459
PFS: First Publisher	69
Ventura Publishing	445

LANGUAGES

Basic Compiler (MS)	$176
C Compiler (MS)	265
Cobol Compiler (MS)	531
Fortran Compiler (MS) 4.1	266
Macro Assembler (MS)	89
Pascal Compiler (MS)	179
Quick Basic	59
Quick C	59
Turbo Basic	62
Turbo C	88
Turbo Pascal 5.0	88
Turbo Pascal Dev. Library	228

GRAPHICS

Designer	$399
Draw Plus	229
Generic CAD w/Dot Plot 3.0	72
Harvard Graphics 2.1	255
In-A-Vision	202
Microsoft Chart 3.0	234
News Room	30
PC Paintbrush Plus	69
Printmaster Plus	28
PrintShop	32
Publishers Paintbrush	132
Turbo Graphix Tool Box	62
VP Graphix	56

UTILITIES

Brooklyn Bridge	$69
Copy II PC	18
Cruise Control	28
Disk Optimizer 4.0	37
Disk Technician Advanced	101
Disk Technician Plus	67
Fastback Plus	98
Mace Utilities	47
Microsoft Windows 286	59
Microsoft Windows 386	115
Norton Utilities 4.5	47
Sidekick Plus	125
Sideways	37
SQZ Plus	53
Take Two MGR	59
XTree Professional	61

DATA BASE MANAGEMENT

Clipper	$405
DB-XL Diamond	107
Nutshell Plus	185
Paradox 2.0	403
PFS: Professional File 2.0	152
Q & A	199
R:Base For DOS	426

MONEY MANAGEMENT

Dollars & Sense w/Forcast	$91
Tobias Manag. Your Money 5.0	114

COMMUNICATIONS

Carbon Copy Plus	$105
CompuServe Subscription Kit	19
Crosstalk XVI	87
Crosstalk Mark IV	109
Mirror III	49
Remote 2	87
Smartcom II	79
Smartcom III	136

INTEGRATIVE SOFTWARE

Ability Plus	$137
Enable 2.0	359
Framework III	469
MS Works	88
PFS: 1st Choice	82
Smart Software System	385
Symphony	469

MICE

PC Mouse w/paint	
(Buss or Serial)	$81
PC Mouse	
w/Autosketch Buss	97
w/Autosketch Serial	97
Microsoft Buss Mouse 1.0	95
Microsoft Serial Mouse 1.0	95
MS Buss Mouse w/CADD	189
MS Buss Mouse w/Windows	125
MS Serial Mouse w/CADD	189
MS Serial Mouse w/Windows	125

SPREADSHEET

Cambridge Analyst	$89
Lotus 1-2-3 Ver. 2.01	285
Lotus Manuscript	289
MS Excel	215
MS Excel/Mach 20 Bundle	629
Multiplan 4.0 Microsoft	116
Quattro	148
Supercalc 5.0	305
VP Planner	47
VP Planner Plus	112

IBM HARDWARE

PRINTERS

ALPS	
Allegro	$345
All Other Models	Call
AST Turbo Lazer	2865
Brother	
M1109	139
LAZER	Call
Other Models	Call
Citizen Printers	
120D	145
180D	165
All Other Models	Call
Diconix	
150	305
Epson	
FX850	355
LQ850	555
LQ500	295
LX800	175
All Other Models	Call
Hewlett-Packard	
Laser Jet Series II	1785
Paint Jet	1085
Desk Jet	725
NEC	
P2200	339
P5200	505
P5300	679
Other Models	Call
Okidata	
320	319
321	449
390	449
New Lazers	Call
All Other Models	Call
Panasonic	
1124	319
1180	185
1191	249
All Other Models	Call
Star Micronics	
NX 1000	169
NX 1000 Rainbow	225
NX 2400	305
All Other Models	Call
Toshiba	
311	385
Pagelazer	2535

MODEMS

Anchor	
1200B w/Software	60
2400EXT	135
2400B w/Software	119
Hayes	
All Models	Call
Prometheus	
1200B w/Software	70
2400B w/Software	125
Other Models	Call
US Robotics	
9600	589
2400	125
All Other Models	Call

NETWORKING

Novell	Call

HAND SCANNERS

Complete	
400	$139
Half Page	175

DISK DRIVES

Bernoulli Box	
10 Meg	$889
20 Meg	1250
Beta Int 20 Meg	765
Hardcards	
Plus 20 MG Hardcard	509
Plus 40 MG Hardcard	645
Passport	Call
Mountain Tape Back Up	
44 MG	340
Seagate	
20 MG w/WD Controller	275
251-1 40 Meg AT	425
40 MG w/Controller	43[illegible]
All Other Models	Call
Teac	
1.2 MG AT	85
Floppy 360K	70
Toshiba	
3.5 Dr 720K	79
3.5 Dr 1.4 MG	99

KEYBOARDS

Keytronics	
KB101 +	$85
KB5151	119

TERMINALS

Wyse	
30	$295
50	378

FAX MACHINES

Brother	
150	$919
Sharp	Call

BOARDS

AST	
Six Pack Plus w/384K	$269
VGA +	329
All Other Models	Call
ATI	
EGA Wonder	225
VGA Wonder	305
Hercules	
Graphics Card Plus	175
All Other Models	Call
Intel	
Inboard 386 PC	720
Co-processors	Call
Aboveboard	Call
Orchid	
Designer 800	235
Pro Designer	295
Pro Designer +	385
Tiny Turbo 286	255
Paradise	
Autoswitch 350	155
Autoswitch 480	170
All Other Models	Call
Video-7	
Fastwrite VGA	345
VGA	265
Vega deluxe	286
Zenith	
449	239

FAX BOARDS

Complete	
4800	$225
9600	389

COMPUTERS

Amstrad	
Dual Drive	$775
Single Drive	775
AST	
Model 80	1430
Other Models	Call
Toshiba	
T-1000	789
T-1200FB	1589
T-1600	3375
T-3200	3895
T-5100	4795
Zenith	
Supersport/2	1655

MONITORS

Amdek Monitors	
210A	$89
410	136
Mitsubishi	
Diamond Scan	489
NEC	
Multisync II	589
Multisync Plus	899
Multisync 3D	Call
Other Models	Call
Princeton	
Ultrasync 12"	489
Ultrasync 16"	839
Other Models	Call
Sony	
Multiscan (1303)	535
Zenith 1490	589

VISA

MasterCard

- No Charge for VISA and Mastercard.
- We Do Not Charge Your Card Until Your Order Is Shipped.
- All products including D.O.A.'s, carry only manufacturer's warranties. We do not honor guarantees, rebates, trial period privileges or promotional programs offered by manufacturers.
- No returns on used or misordered product.
- You Pay The Ground Shipping 1-10# $6.00; 11-20# $10.00 (Except Alaska & Hawaii), We Pay the Air Difference.

- Free Air Applies ONLY to Orders Over $100.
- No APO, FPO, or International orders please.
- Call Before Submitting P.O.'s. Ask for National Accounts.
- Personal and Company Checks Will Delay Shipping 3 Weeks.
- Prices, Terms & Availability Subject to Change Without Notice.
- Add 5% for C.O.D. Orders.
- We Do Not Guarantee Machine Compatibility.
- Mailing Address: TELEMART, 8804 N. 23rd Ave., Phoenix, AZ 85021.

- To Place an Order: 1-800-834-6270.
- To Follow Up On An Order: (602) 944-1037.
- Order Line Hours: Mon.-Fri. 7:00 am-6:00 pm Saturday 9:00 a.m.-1:00 p.m.
- Order Processing: (602) 944-1037 10:00 a.m.-3:00 p.m. Mon.-Fri.

MMC

ments definition, and finally systems planning and architecture.

At a conceptual level, DSSD still retains features that characterized it at the programming level. For example, it still focuses (in its design phase) on working backward from outputs. But at the systems level, instead of working backward to the ideal inputs, as it does in the programming methodology, DSSD works backward to the logical database and then to the inputs (see figure 1). The logical database turns out, not surprisingly, to be a normalized relational database.

While a complete definition of the results (outputs plus algorithms) is an excellent point at which to begin the design process, it is not the proper place to start requirements definition. So, over the years, DSSD has been extended to cover first the context, then the functions, and finally the results of the system in question.

Thus, a number of tools were needed to facilitate this process. *Entity diagrams* help you define the systems context, and *assembly-line diagrams* (a modified form of Warnier/Orr diagrams) help you define the functional flow of the system.

Data-structured methodologies have, I believe, a leg up on more process-oriented methodologies, since they are more rigorous and hence provide a better basis for true integration throughout the systems life cycle. DSSD has been used successfully on a range of software systems, from commercial on-line systems to real-time control systems. Thousands of people have been trained and thousands of systems have been built using it.

DSSD is a software engineering approach that has provided a stable framework for incorporating new technologies as they come along. For example, we have incorporated prototyping, on-line, and real-time design into DSSD without sacrificing the rigor or completeness. But there is a catch: To use DSSD successfully, you must invest time in training, use, and automation. In software engineering, as in life, there is no free lunch.

BIBLIOGRAPHY

Hansen, Kirk. *Data Structured Program Design*, 2d ed. Englewood Cliffs, NJ: Prentice-Hall, 1986.

Higgins, David A. *Designing Structured Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

Jackson, Michael A. *Principles of Program Design*. New York: Academic Press, 1975.

Orr, Kenneth T. *Structured Systems Development*. New York: Yourdon Press, 1977.

______. *Structured Requirements Definition*. Topeka, KS: Ken Orr & Associates, 1980.

Warnier, Jean-Dominique. *Logical Construction of Programs*. New York: Van Nostrand Reinhold, 1976.

______. *Program Modification*. Boston: Martinus Nijhoff, 1978.

______. *Logical Construction of Systems*. New York: Van Nostrand Reinhold, 1981.

Ken Orr is president and chief scientist for Optima in Schaumburg, Illinois, and a principal of the Rosetta Institute in Topeka, Kansas. He is a frequent lecturer and the author of several books and articles on programming, systems and database design, and CASE. He can be reached on BIX c/o "editors."

The Gane/Sarson Approach

Chris Gane

When we think about an information system that doesn't exist yet, our ideas are usually pretty vague and general. This is not an accusation; it's a fact of human psychology.

The purpose of logical modeling is to take these necessarily vague ideas about requirements and convert them into precise definitions as fast as possible. Part of the speed comes from having graphical techniques that enable you to put down the essence of a system without going through the trouble of actually physically implementing it, as you might do, for example, in a prototype.

Several approaches to logical modeling have been proposed. The one outlined here is the current version of the approach set out in a book I wrote with Trish Sarson (see reference 1). It has become generally known as the Gane/Sarson methodology.

Logical Modeling

You can think of logical modeling as a seven-step process. Suppose the users say, "We need a system that integrates sales, inventory control, and purchasing." What exactly does that mean?

• *Step 1.* Develop a system-wide data-flow diagram (DFD) describing the underlying nature of what occurs in the sales, inventory control, and purchasing areas of the business. The simplicity of the DFD comes from the use of only four symbols to produce a picture of the underlying logical nature of any information system, at any desired level of detail.

Figure 1 shows CUSTOMERS (an external entity, something outside the system) sending in a stream of sales orders along the data-flow arrow. Process 1, process sales, handles those orders using product information from the data store called D1: PRODUCTS and puts information about sales into the data store named D3: SALES.

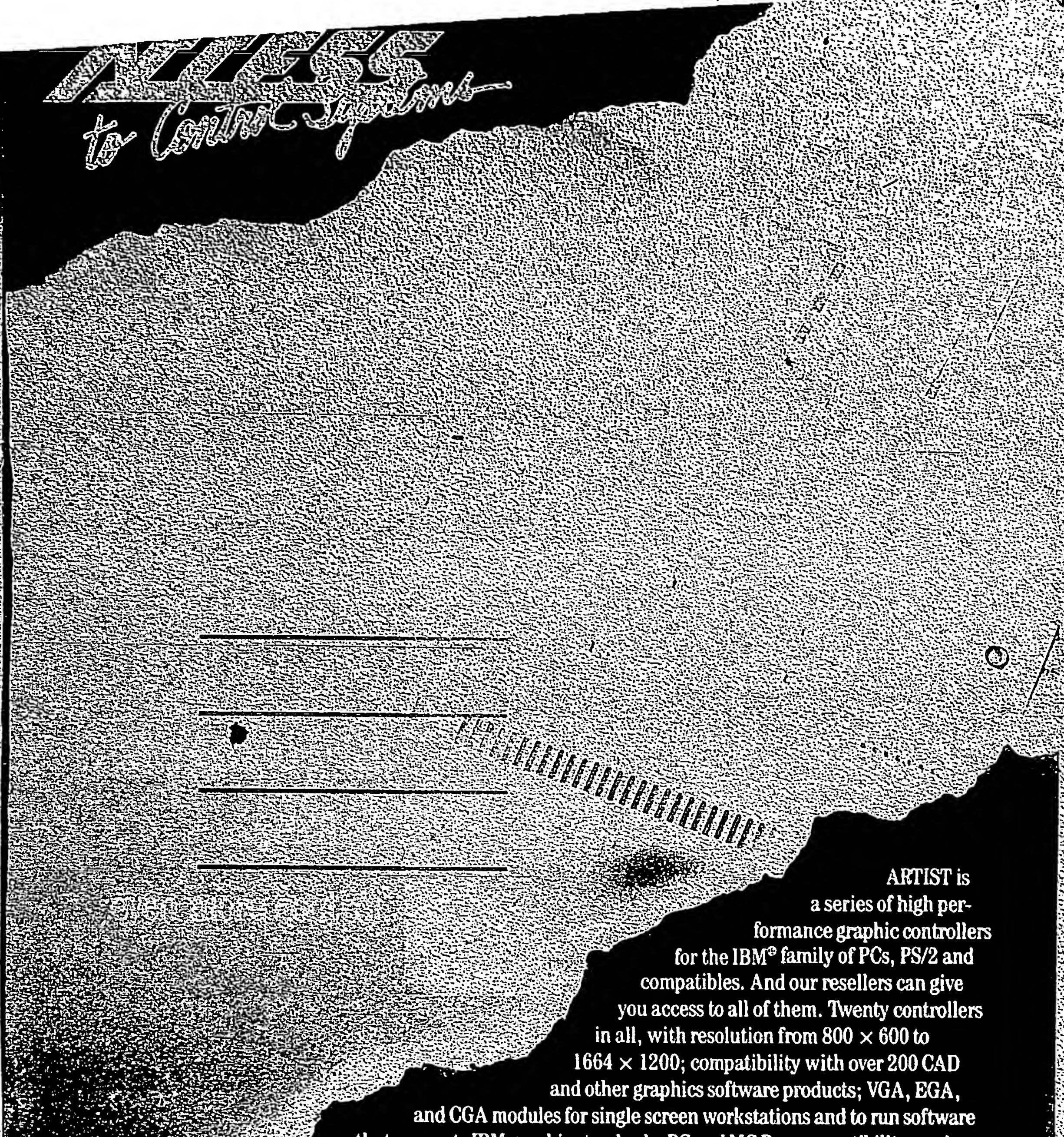
This figure also shows the whole of the business area, depicted using only the four symbols. For each sale, process 1 updates the INVENTORY data store, D2, with the units sold. The data stored in D3 is used by processes 2 and 3 to prepare bank deposit documents and send them to the bank, and to prepare sales reports and send them to management.

At some appropriate time—notice that time is not shown on the DFD—process 4 extracts information about the inventory status of various products from D2 and combines it with information from D3 concerning their past sales, to determine whether a product needs to be reordered. If so, based on information in D4, which describes the prices and delivery times quoted by suppliers, process 4 chooses the best supplier to order from.

Process 4 sends purchase orders to the external entity SUPPLIERS and stores information about each purchase order in D5: POS_IN_PROGRESS. When a shipment is received from a supplier, process 5 analyzes it, extracting data from POS_IN_PROGRESS to determine whether what has been received is what

continued

ACCESS to Control Systems



ARTIST is a series of high performance graphic controllers for the IBM® family of PCs, PS/2 and compatibles. And our resellers can give you access to all of them. Twenty controllers in all, with resolution from 800 × 600 to 1664 × 1200; compatibility with over 200 CAD and other graphics software products; VGA, EGA, and CGA modules for single screen workstations and to run software that supports IBM graphic standards; PC and MC Bus compatibility; and developer toolkits for PGL, DGIS, X-Windows and Hoops.

Best of all, when you buy through an ACCESS reseller, you're assured that our knowledge and experience is behind the sale. Because ACCESS is the only national graphics equipment distributor with on-going training, performance guarantees and reseller support. So why buy from just anyone, when you can buy from a reseller with ACCESS to graphics technology.

Control Systems™ 1-800-826-4281. In Canada: 1-800-543-6523

ACCESS Graphics Technology — The National Graphics Distributor.
For resellers in your area call 1-800-443-4223

Circle 316 on Reader Service Card

was ordered, incrementing the INVENTORY, D2, with the accepted amount, and storing the accepted quantities in POS_IN_PROGRESS.

This DFD achieves three things. First, it sets a boundary to the area of the system and of the business covered by the system. Things represented by the external-entity symbol (i.e., customers, the bank, management, and suppliers) are, by definition, outside the system. Processes not shown are not part of the project. For example, the diagram shows receiving shipments from suppliers but not handling the invoices received from them, implying that accounts payable is outside the scope of the project as well.

Second, it is nontechnical. Nothing is shown on a DFD that is not easily understandable to people familiar with the business area depicted, whether or not they know anything about computers.

Third, it shows both the data stored in the system and the processes that transform that data. It shows the relationships between the data and the processes in the system.

∘ *Step 2.* Derive a first-cut data model—that is, a list of the data elements to be stored in each data store, as defined on the DFD. You should draw up this list from your own knowledge and from the knowledge of users about what information you need to describe a product, a supplier, a sale, and so on.

You can refine the list by looking at each system input, such as sales orders or shipments in figure 1, determining what data elements each input represents, looking at each output in the same way, and then working from the outputs back to the data stores or from the inputs forward to the data stores.

∘ *Step 3.* See what entity-relationship analysis can tell you about the structure of the data to be stored in the system. First, you ask, "What are the entities of interest about which I may need to store data?" For this business, the answer might be CUSTOMERS, PRODUCTS, INVENTORY, SUPPLIERS, SALES, and PURCHASE_ORDERS. Then, you create a diagram with a block for each entity you have identified. (It is conventional in this diagram to state the entities as singular nouns—for example, CUSTOMER instead of CUSTOMERS.)

Next, looking at each pair of entities on the diagram, you ask, "What, if any, relationships exist between them?" For example, you know that one customer may be associated with many sales, but each sale can be for only one customer. This is conventionally shown by a line with an arrowhead against the "many" block and a plain line at the "one" block.

Take, for instance, PRODUCT and SALE: One product may be associated with many sales, and one sale may be for many products—at least one, and possi-

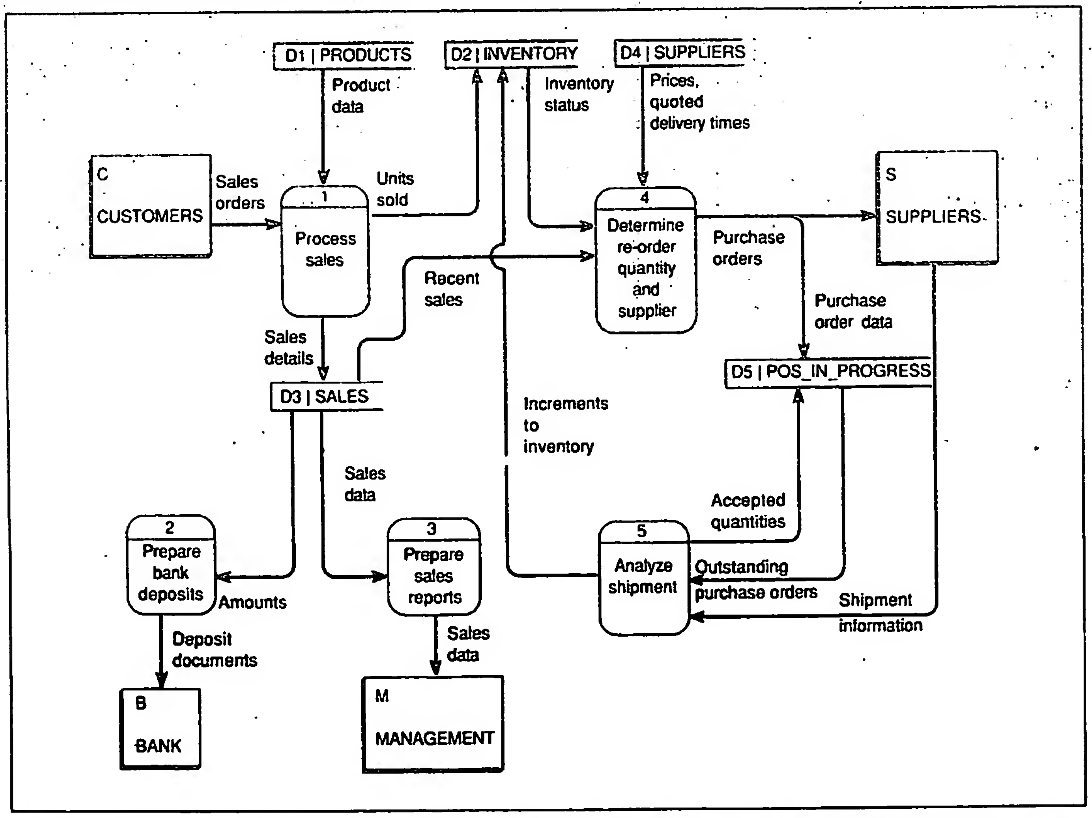


Figure 1: *A DFD for the whole of the business area. Note the box for external entities, the open rectangles for data stores, the rounded box for the process, and the data-flow arrow, which shows the direction of data movement. Notice also that time is not shown on a DFD.*

bly more. This relationship is shown by a line with an arrowhead on both ends. On the other hand, each product has only one inventory record, and each inventory record refers to only one product. Consequently, they are joined by a simple line. Adding in all the identifiable relationships creates a diagram like figure 2.

◦ *Step 4.* Use all the information you have about the data so far to describe the data model as one made of linked, two-dimensional tables. These tables should be *normalized* (i.e., made as simple as possible). One way to summarize the rules of normalization is to say that in a properly simplified table, in which a column or combination of columns uniquely identifies each row (the key), each nonkey column should depend only on the key.

◦ *Step 5.* Redraft the DFD to reflect a more precise view of the system data as a result of entity-relationship analysis and normalization.

◦ *Step 6.* Partition this logical model of process and data into *procedure units*—that is, chunks of automated and manual procedures that can be executed (and therefore developed) as units. To do this, you consider each input and output and ask the following questions for each one:

1. When does it happen?
2. How large an area of the DFD is involved in handling or producing it?
3. Can that area be implemented as a single unit? If not, why not?

◦ *Step 7.* Specify the details of each procedure unit that will be required to implement the system. A procedure-unit specification may involve

1. an extract from the system DFD showing where this procedure unit fits into the rest of the system;
2. details of the tables accessed by the procedure unit;
3. layouts for any screens and reports involved in the procedure unit; and
4. details of the logic and procedures to be implemented, written in structured English or some other unambiguous form.

With the nature of the procedure unit defined, you can decide whether it should be prototyped or implemented directly in the target language. You can develop the screen and report layouts by prototyping.

Steps 6 and 7 in this sequence are not, strictly speaking, logical modeling, since they deal with converting the logical model into a physical model. They are included, however, because they form part of the natural flow of thought processes beginning with defining the system and ending in its physical design.

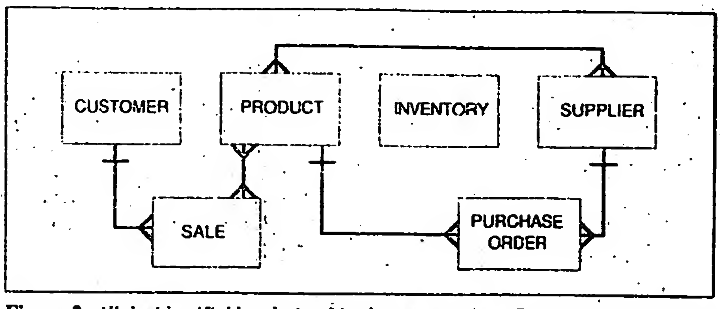


Figure 2: *All the identifiable relationships between entities. For each pair of entities on the diagram that has a relationship between its elements, the relationship may be one-to-one, one-to-many, many-to-one, or many-to-many. This diagram provides a lot of information about the system in showing all the relationships that exist between the entities involved.*

Editor's note: *Chris Gane extracted this article from Chapter 1 of his book* Rapid System Development, *published by Prentice-Hall in December 1988.*

REFERENCES
1. Gane, Chris, and Trish Sarson. *Structured Systems Analysis.* Englewood Cliffs, NJ: Prentice-Hall, 1979.

Chris Gane is president of Rapid System Development in New York City and principal consultant at Bachman Information Systems in Cambridge, Massachusetts. He is the author of several books, including Rapid System Development *(Prentice-Hall, 1988). He can be reached on BIX c/o "editors."*

The Yourdon Approach

Edward Yourdon

The Yourdon method is a generic, ecumenical collection of software engineering ideas developed over the past 20 years by a variety of people who have worked at Yourdon, Inc. Taken together, these ideas are often referred to as *structured techniques*: structured programming, structured design, and structured analysis.

Because of the continuing influx of new ideas from new people, the Yourdon method is constantly evolving. The method that thousands read about in Tom DeMarco's book in 1978 (see reference 1) has changed considerably in the past 10 years. And the Yourdon method of 1989 is evolving to incorporate the best ideas of object-oriented design and analysis.

But what *is* the "Yourdon method" today? It consists of two things: tools and techniques. The tools are a variety of graphical diagrams used to model the requirements and the architecture of an information system. The most familiar of these tools is the data-flow diagram (DFD) (see figure 1). The original DFD notation was extended a few years ago to support real-time systems; a real-time DFD includes control flows and control processes. For a detailed description of

continued

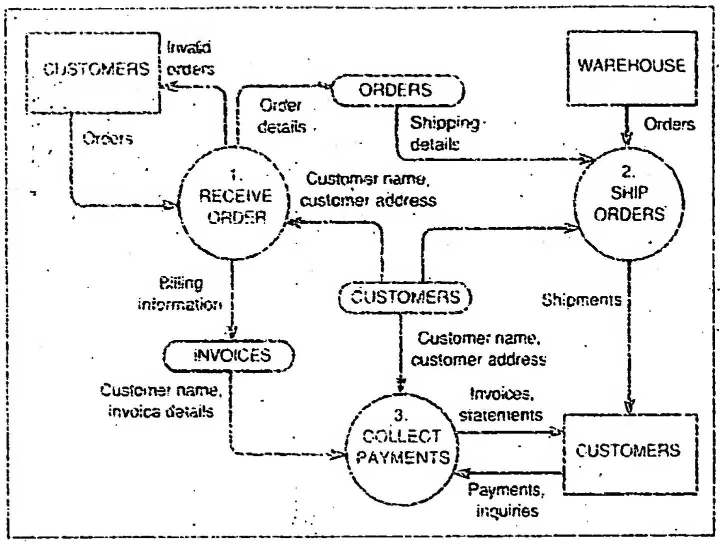


Figure 1: *A data-flow diagram. The DFD models the functions that a system must perform.*

real-time DFDs, see reference 2.

While the DFD is an excellent tool for modeling the *functions* that a system must carry out, it says little or nothing about data relationships and time-dependent behavior. Thus, the current Yourdon method also includes entity-relationship diagrams (ERDs) and state-transition diagrams (STDs) (see reference 3).

After you have finished describing the system requirements, you can use a structure chart to illustrate the organization of modules that will implement those requirements. A number of guidelines exist that the systems analyst can follow to ensure that each diagram is complete and logically consistent.

While the graphical diagrams provide an effective way of communicating information about different aspects of a system, they don't tell the whole story. For a complete system description, you need additional textual support: a data dictionary, which describes the composition of each data element, and a set of process specifications that describe the required behavior of each bottom-level "bubble" in the DFD.

The Techniques

The techniques of the Yourdon method consist of some "cookbook" guidelines that help you go from a blank sheet of paper, or a blank screen, to a well-organized system model. Originally, these guidelines were based on the simple concept of top-down partitioning of system functions (e.g., draw a single bubble or box to represent the entire system, then draw lower-level bubbles or boxes to represent subsystems, and so forth).

Today, the Yourdon method uses a technique known as *event partitioning* (see reference 4). This approach begins by drawing a top-level *context diagram* to identify the system boundary and to define the interfaces between the system and external sources and sinks. Then, after interviewing the user, you can write a list of the *events* that occur in the external environment and to which the system must respond. (Events are often input transactions.)

The event-partitioning approach provides a simple guideline to help you compose a first-cut crude DFD: For each event, draw one bubble whose function is to provide the required response to the event. (In most cases, the response involves generating an output, but it may also involve storing some information in a data store to be used by some subsequent event.)

For a system with 100 events, the DFD would have 100 bubbles. This is too complex to work with, so the event-partitioning technique provides guidelines to help you partition *upward*—that is, to gather several of the DFD bubbles together and represent them by a single bubble in a higher-level DFD. The strategy for deciding which bubbles should be grouped together is to look for bubbles that deal with common data (e.g., a common data store). In this sense, event partitioning is very similar to the object-oriented design approach.

There are various additional guidelines and techniques to help you compose well-formed models of both system requirements and system architecture. (One book that discusses both the analysis area and the design area—as well as the "twilight zone" that separates the two—is given in reference 5.)

The Yourdon Philosophy

Throughout all of the Yourdon method—regardless of variant or dialect, whether you draw circles or ovals in your DFD, or where you hear about it—you will see the following philosophies.

• *Modeling is good.* Developing a model of a system before you build it is almost always a useful, educational activity. For this to work, however, the model has to be inexpensive and easy to build: If it costs as much to develop the model as to develop the system, it's obviously a waste of time. The model also has to be accurate—it should not mislead you or lie to you. And it should be easy to understand: It should highlight those aspects of the system that are important, and it should deemphasize or hide those aspects that are unimportant or uninteresting.

Since most systems are complex in three different dimensions—functions, data, and timing and control—it is useful to have three different types of models, DFDs, ERDs, and STDs, each of which illustrates a single perspective of the system. The Yourdon method is based on abstract, pictorial models—either on paper or on a computer screen.

Another approach is to develop a prototype of the system as a model—a living, breathing model instead of a passive collection of diagrams. When prototyping was first introduced in the early 1980s, it was considered an alternative to paper-based modeling approaches—the systems analyst was often told to make a binary choice between prototyping and drawing DFDs.

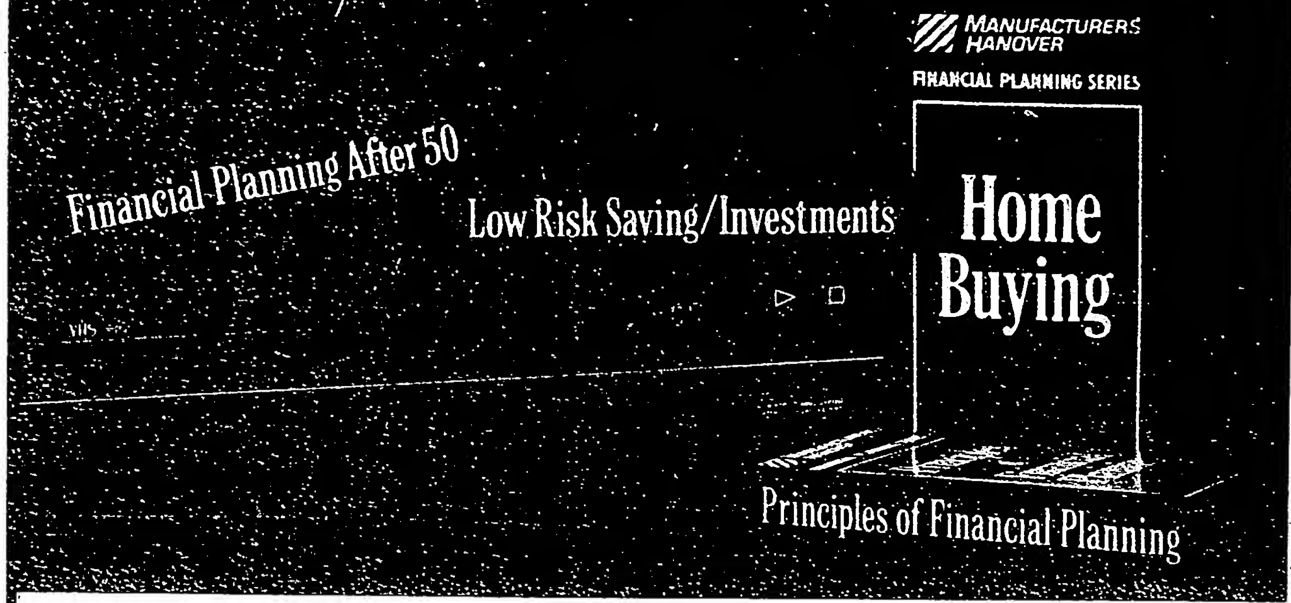
Today, we know that the two approaches are complementary: You can draw diagrams as a permanent record of system requirements and use prototyping to experiment with such key issues as the user interface (input screens, report layouts, and so on). For a good discussion of the marriage of prototyping and "classical" structured analysis, see reference 6.

• *Iteration is good.* As fallible humans with limited intelligence, we rarely, if

continued

MANUFACTURERS HANOVER

INTRODUCING 4 NEW WAYS TO AVOID MAKING COSTLY FINANCIAL MISTAKES.


MANUFACTURERS HANOVER
FINANCIAL PLANNING SERIES
Home Buying
Low Risk Saving/Investments
Financial Planning After 50
Principles of Financial Planning
VHS

MHT's Financial Planning VHS Tapes. Buy 2, Get 1 Free!

These tapes offer four smart ways to avoid making costly financial mistakes. Developed by a Certified Financial Planner, each is filled with valuable money-saving ideas. They're smart to give, smart to get.

PRINCIPLES OF FINANCIAL PLANNING

Kathleen Covey, CFP. Prevent costly errors by learning:

- how to evaluate your savings and investments needs and options, how to diversify your investments and protect yourself from economic cycles.
- how to prevent inflation and taxes from eroding your savings and investment returns.
- how much life, disability, property, and liability insurance you may need, and what to look out for.

LOW-RISK SAVINGS/ INVESTMENTS

Charles Lefkowitz, CFP. Assess investment risks by learning:

- why tax-exempt savings and investments could end up costing you money.
- how Unit Investment Trusts, Mutual Funds, Mortgage-Backed Securities, Annuities, Tigers, Strips, and Zero Coupons may provide higher returns than what you're now getting.
- how you could lose your shirt on bonds because their ratings do not tell the whole story. How "Call Features," interest changes, and other factors could ruin your rate of return.

FINANCIAL PLANNING AFTER 50

Arden D. Down, CFP. Secure your financial future by learning:

- how to evaluate your company pension/profit-sharing plan so that you won't be surprised at retirement.
- how to design an asset-allocation investment plan to minimize risk and incorporate tax-advantaged strategies for future income needs.
- how to judge which makes the best tax sense: lump sum, rollover, or averaging your pension distribution.

HOME BUYING

Kathleen Covey, CFP, and Arden D. Down, CFP. Learn all aspects of the home-buying process:

- what are the differences between assessed value and appraised value: appraised value versus market value? Not knowing might cost you.
- learn more about the "hidden" costs of owning a house, a co-op, or condo. Closing and other costs are often underestimated. Buyers beware.
- which is better, a mortgage with points or without? What's deductible under the new tax law?
- what are ARMS, CAPS, amortization, escrow, titles, private mortgage insurance, etc.? Understanding the jargon could save you money.

Call now to order your gift.

1-800-533-0552

Our operators are standing by 24 hours a day, 7 days a week. Or, complete the coupons below and mail to:

Manufacturers Hanover
P.O. Box 3769, N.Y., N.Y. 10163

Circle 333 on Reader Service Card

Titles	Send Me (qty)	Cost	Total Cost
PRINCIPLES OF FINANCIAL PLANNING		X $39.95	
LOW-RISK SAVINGS INVESTMENTS		X $39.95	
FINANCIAL PLANNING AFTER 50		X $39.95	
HOME BUYING		X $39.95	
		SUBTOTAL	
		NY residents add local sales tax	
		$7.50 per tape, shipping and handling	
		TOTAL	

[ ] My check is enclosed Charge to Visa [ ] MasterCard [ ]
Make check payable to Manufacturers Hanover Trust

Credit Card #

Signature Exp. Date

Social Security #

Indicate your choice of one free tape:

☐ PRINCIPLES OF FINANCIAL PLANNING ☐ FINANCIAL PLANNING AFTER 50
☐ LOW-RISK SAVINGS/ INVESTMENTS ☐ HOME BUYING

NOTE: Limit one free tape per order.

Send my order to

My Name

Address

City State ZIP

( )
Business Home Phone

Offer expires 12/31/89. BY-4/89

ever, develop a perfect solution to a complex problem on the first attempt. At best, we can hope for a crude beginning that, through iteration, we can gradually refine and improve. To practice iteration, we must have models that are easy to create and easy to revise. In the past, we grappled with the finality of pen and ink; with the word processors of today, most of us take iteration for granted in composing reports and memos.

In systems development, we tried to make iteration of system models easier by insisting on partitioning the overall system model into a number of separable submodels. Thus, if one aspect of a system changes, ideally only one page of a diagram has to be modified. As a practical matter, though, most systems analysts in the 1970s and early 1980s drew DFDs only once—on paper. This is one reason why today's microcomputer-based computer-aided software engineering products are so important: They make iteration a practical reality.

• *Partitioning is good.* When we first learned how to write computer programs, we were given simple problems that we could finish in a day or two, keeping every aspect of the problem in our heads at once. With real-world programming problems, however, the only way we can successfully build systems that, today, typically involve more than a million lines of code is by partitioning the system into smaller and smaller pieces.

There are great debates about whether the partitioning should be based on functional decomposition or data decomposition. But either approach, followed rigorously, is better than no partitioning, or sloppy partitioning that leads to subsystems with subtle, pathological interconnections.

ACKNOWLEDGMENTS
The author gratefully acknowledges the contributions of the following people to the Yourdon method: Tom DeMarco, Larry Constantine, Chris Gane, Trish Sarson, Steve McMenamin, Tim Lister, John Palmer, Paul Ward, Steve Mellor, Meilir Page-Jones, Bob Block, Al Brill, Tim Wells, and Matt Flavin.

REFERENCES
1. DeMarco, Tom. *Structured Analysis and System Specification*: New York: Yourdon Press, 1978.
2. Ward, Paul, and Steve Mellor. *Structured Techniques for Real-Time Systems*. New York: Yourdon Press/Prentice-Hall, 1985.
3. Yourdon, Edward. *Modern Structured Analysis*. New York: Yourdon Press/Prentice-Hall, 1989.
4. ———. *Essential Systems Analysis*. New York: Yourdon Press/Prentice-Hall, 1984.
5. Page-Jones, Meilir. *The Practical Guide to Structured Systems Design*, 2d ed. New York: Yourdon Press/Prentice-Hall, 1988.
6. Connell, John, and Linda Schafer. *Rapid Prototyping*. New York: Yourdon Press/Prentice-Hall, 1989.

Edward Yourdon is an independent consultant and the publisher of a software engineering newsletter, American Programmer. *He is an author in the field of structured systems development. His latest book is* Modern Structured Analysis *(Yourdon Press/Prentice-Hall, 1989). He can be reached on BIX c/o "editors."*

The Entity-Relationship Approach

Peter P. Chen

One of the major problems in software engineering today is the piecemeal approach to systems design. This approach makes the integration of different application systems difficult, if not impossible. We try to design the data structures and formats to fit current processing needs and then later run into problems of data conversion and integration.

An integrated database is a solution to these problems. However, acquiring a DBMS does not make them go away. What is needed is a structured methodology that can systematically convert user requirements into well-designed databases. The entity-relationship (ER) approach is such a methodology.

Let's start with an example. Say you need a program to keep track of the list of employees working for each department in your company. This program needs to accept data on the screen, store it on disk, and print out the report on demand. The programmer/analyst comes up with a file format (see figure 1a).

In the meantime, another group in the company implements a program to keep track of employee information for each project; the file format in this program turns out a little different from the other (see figure 1b). Each program satisfies the needs of the group that requested it. However, one day the company president wants to know which departments have employees working on project X. Then everyone scrambles around trying to convert the data in one file to the format of the other file. Let's look deeply into these two file formats to see what kinds of problems they had.

- Synonym (the same data element has different names): For example, SOC_SEC_NO in figure 1a is the same data element as SS# in figure 1b.
- The same name for different data elements: For example, NAME in figure 1a refers to the name of an employee, while NAME in figure 1b is the name of a project.
- Incompatibility of data formats: For example, the data-type format of AGE in figure 1a is Int(2), while the data format of AGE in figure 1b is Real(3.2).
- Duplication of data: For example, the project data (PROJ#, NAME) is duplicated for each employee associated with the project in figure 1b, and the BUDGET data of each department in figure 1a is repeated for each employee.
- Update anomalies: For example, changing any of an employee's data-element values in one file but not in the other will result in inconsistent data.

If the above file designs are not good, what would be a good design? How many record types (or relations in the case of relational databases) should there be? Should there be one huge record type consisting of all data elements, or the other extreme—many small records, each consisting of a pair of data elements? Furthermore, what is the primary key for each record (relation) type? The main question is: Do we have a methodology for file and database design? The answer is yes, and the leading methodology is the ER approach.

Six years ago, a survey of Fortune 500 companies (published in ACM SIGMOD proceedings, 1983) conducted by two Ohio professors showed that the ER methodology ranked as the most popular methodology in data modeling and database design. Why? Because it is simple, easy to understand by noncomputer people, and theoretically sound. To illustrate, here are the major steps of the ER approach using the above example:

• *Develop an entity-relationship diagram (ERD).* This step identifies ER types and associated attributes and also the primary keys for each entity type.

An *entity* is a thing (e.g., a person or an automobile), a concept, an organization, or an event of interest to the organization doing the modeling. An *entity type* is a classification of entities satisfying certain criteria. A *relationship* is an interaction between entities. A *relationship type* is a classification of relationships based on certain criteria. Usually, nouns in English correspond to entities, while verbs correspond to relationships.

In the example in figures 1a and 1b, you can identify three entity types: DEPT, EMP, and PROJ. You can also identify two relationship types: HAS and WORK_FOR (note that relationship-type names are verbs). Figure 2 depicts an ERD in which rectangular boxes represent entity types and diamond-shaped boxes represent relationship types.

The next step is to identify the *cardinality* of the relationship types. The cardinality of HAS (between DEPT and EMP) is 1:*n* (one-to-many); that means a department can have many employees, but each belongs to at most one department. The cardinality of WORK_FOR (between EMP and PROJ) is *n*:*n* (many-to-many). You then identify the properties (attributes) of each ER and express them graphically as circles (or ellipses). For example, each DEPT has attributes DEPT# and BUDGET. The primary key is indicated by a double circle. Note that there is an attribute called %TIME for relationship WORK_FOR.

• *Convert the ERD into conventional file and database structures.* There are rules for doing this. For example, you can convert the ERD in figure 2 into the relational structure with all the primary keys underlined.

DEPT(<u>DEPT#</u>, BUDGET)
EMP(<u>SS#</u>, NAME, AGE, DEPT#)
PROJECT (<u>PROJ#</u>, PNAME)
WORK_FOR(<u>SS#</u>, <u>PROJ#</u>, %TIME)

Simply speaking, each entity type is converted into a relation, and a relationship type is converted into a stand-alone relation or consolidated with another relation, depending on the cardinality of the relationship.

If you are familiar with relational normalization theory, you can prove that these relations are in Third Normal Form. As you can see, all the primary keys of the relations are derived automatically; and DEPT# in EMP relation is a *foreign* key (i.e., the primary key of another relation—DEPT).

• *Develop application programs based on the file and database structures.* If you are using a relational DBMS, you can now write a System Query Language (SQL) program to express the question, Which departments have employees working on project X?

```
SELECT EMP.DEPT#
FROM EMP, WORK_FOR
WHERE (WORK_FOR.PROJ# = X)
AND(WORK_FOR.SS# = EMP.SS#)
```

This article shows how to design a relational database based on the ER approach. Similarly, you can design file structures and various other databases—from microcomputer-based DBMSes,

continued

a)

Layout	SOC_SEC_NO	NAME	AGE	DEPT#	BUDGET
Format	Char(9)	Char(20)	Int(2)	Int(3)	Int(9)

b)

Layout	SS#	EMP_NAME	AGE	PROJ#	NAME	%TIME
Format	Char(9)	Char(20)	Real(3.2)	Int(4)	Char(30)	Int(2)

Figure 1: **(a)** *File format for the program to list employees in each department.* **(b)** *File format for the program to list projects for each employee.*

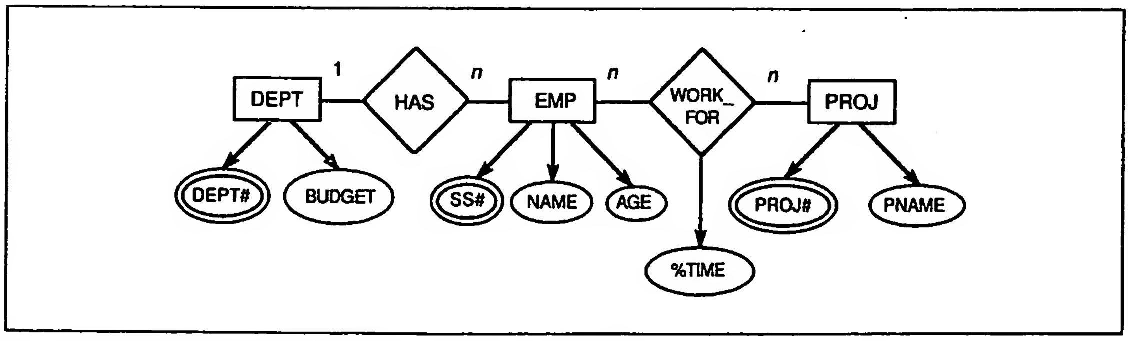


Figure 2: *Entity-relationship diagram (ERD) for a database based on figure 1.*

such as dBASE, to mainframe-based DBMSes, such as DB2 and IMS—based on the ER approach.

Future Trends

You have seen how to design a database and an application program based on the ER approach. The resultant database is sound and avoids such problems as data duplication and update anomalies. Commercial tools are available today to automate the ER approach.

The ER model can be used not only as a design tool but also as the underlying model for a DBMS. In the microcomputer and minicomputer range, Zanthe (Ottawa) has a product called ZIM. In the mainframe area, several computer vendors have ER-like DBMSes ready for marketing. For example, Software AG has ADABAS/Entire, and Unisys has SIM as part of its InfoExec offering.

On another front, ANSI recently approved an Information Resource Dictionary Systems standard based on the ER model. In the near future, we'll see a flood of IRDS products as well as computer-aided software engineering tools based on the ER model.

BIBLIOGRAPHY

ANSI. *Standards on Information Resource Dictionary Systems*, 1988.

Chen, Peter P. "Entity-Relationship Model: Toward a Unified View of Data." *ACM Transaction on Database Systems*, vol. 1, no. 1.

Davis, C. G., S. Jajodia, P. Ng, and R. Yeh, eds. *Entity-Relationship Approach to Software Engineering*. New York: North-Holland, 1983.

March, Sal, ed. *Entity-Relationship Approach*. New York: North-Holland, 1988.

Teorey, Toby, and James Fry. "An Extended Entity-Relationship Approach to Logical Database Design." *ACM Survey*, 1986.

Peter P. Chen is Foster Distinguished Chair professor at Louisiana State University and the founder of Chen & Associates, both in Baton Rouge, Louisiana. He received a Ph.D. in computer science/applied mathematics from Harvard University. He can be reached on BIX c/o "editors."

The Structured-Design Approach

Larry L. Constantine

The computer field likes big words. Why call something an *instance* when *instantiation* works just as well, even if it isn't in the dictionary? A software design *method* sounds like the sort of generic-brand thinking that anyone could work out over a long weekend. But a software design *methodology* sounds like an elaborate and well-thought-out concept, perhaps worth attending a seminar on by a major software guru, and certainly worth the price of a book. However, *methodology* actually means the study of methods, and *software methodology* is an ungrammatical use of the word.

Structured design is both a generic term for various systematic approaches to designing program structure and also a kind of brand name for one particular approach. The structured design world is a competitive arena. Varying principles or specialized diagrams reflect some real technical differences. But, more than anything else, competing approaches are based on product differentiation, personal ego, and the territorial imperative. These methods, their associated tools, and the names of the principals are widely recognized.

The school with the longest legitimate claim to the banner of structured design is the Constantine-Myers-Stevens-Yourdon (in alphabetical order, of course) approach that I originated in the late 1960s. It begins with a data-flow diagram (DFD) (often called a "bubble chart") showing the transformational structure of an information-processing problem; then it derives a model of the modular structure of software that will solve that problem.

Models

Much is made of design methods, but structured design is really powered by a *troika* consisting of models, methods, and measures (see figure 1). The models make it possible to picture and play with the modular structure of software systems without actually having to program them first.

System-structure modeling, now accepted as essential to software engineering, was a novel and suspect notion when I first introduced it. The models used in structured design are graphical tools, annotated to represent the structure of problems and programs. For example, the structure chart, an elaboration of the older hierarchy chart, shows all the modules in a system and their essential interrelationships in one compact model. It allows you to see the "shape of things to come" and to explore alternative ways to organize software.

Measures

Structured design, unlike some other structured techniques and software engineering "methodologies," is grounded in a body of underlying theory about what makes programs complicated to build right in the first place and difficult to change in the second. The practical embodiment of this theory takes the form of two measures—coupling and cohesion—that index the relative complexity or difficulty of various designs.

Simple programs are, simply put, made out of little pieces, each of which is easily thought of as a unit or a whole that is mostly independent of other pieces. Module cohesion is a measure of module "wholeness," and coupling measures interdependence. In other words, good designs that are easy to build and change are based on a bunch of modules, each of which is "cohesive," or well-glued together, and only loosely "coupled" to other modules.

Designing in this way, you can program very large systems by writing only small, separate pieces of code. This theory is proving to be the most durable element of the troika. A decade of research has demonstrated the soundness of the basic assumptions about coupling and cohesion and has refined our understanding of how they affect programming and maintenance costs, fault rates, and ease of modification. Quantitative metrics based on the theory now make it possible to automate design evaluation and even parts of the design process.

Object-oriented methods are emerging as major factors in software engineering, but even with these powerful new techniques, the cohesion of actual modules that implement object classes and their coupling with other modules turn out to be important for building simple systems with truly reusable components. As a consequence, the original measures of coupling and cohesion have now been extended and adapted to evaluate the quality of so-called object class modules and abstract data types.

Methods

Methods is the weak third member of the structured design team, but the one that gets much of the attention. The methods of structured design include a loose collection of rules and some moderately systematic strategies for the step-by-step design of software. These strategies are based on specific kinds of software organization that have proved effective in practice. Among the most durable are the balanced system structure known as *transform-centered* organization and the event-oriented organization called *transaction-centered*. Distinct design methods are aimed at producing each variation on system organization.

You might outline an overall structured design method as follows:

1. Develop a nonprocedural (method-independent) statement of the system requirements, usually centered around a DFD.
2. Based on the structure of the problem, choose an appropriate software organizational model or combination.
3. Guided by the data flow and the chosen organizational model, decompose overall functions into subfunctions and compose primitive functions into higher-level functions until the complete requirements are satisfied.
4. Using various design rules and measures of coupling and cohesion, refine the design for increased modularity, extensibility, and likely reusability of modules.
5. Complete detailed designs as necessary for all modules.

The problem, of course, is that actually carrying out the modeling, evaluation, and refinement involved in structured design takes discipline and time, uses a lot of paper, and can wear out the erasers on every pencil in the office. Unaided by computer tools, many developers who use structured design have used it informally and unsystematically, while others have used its concepts to shape their thinking without ever applying the formal models, measures, and methods.

Automating Methods

Computer-aided software engineering is not a replacement or substitute for systematic methods; it's the key that unlocks their full potential. With CASE tools, real structured design on problems of interesting size becomes truly practicable for the first time. The diagram editors of CASE systems assist in developing and refining complex graphical models. CASE tools with built-in knowledge of the models' meaning can check them for consistency and conformance with the established rules of structured design.

With intelligence about good designs incorporated into CASE tools, the computer can evaluate the quality of actual structured designs or even sketch out a rough initial design. At present, the most advanced CASE workbenches provide powerful support for existing software engineering methods; in the future, new structured methods are likely to be developed based on the use of quasi-intelligent CASE tools.

CASE systems can also reinforce or impose standards for software architectures and the diagrams that document them, but CASE vendors have yet to take on much responsibility for standardization. Most of the available tools and workbenches are methodology-independent—meaning you can use them to do any brand of design, structured or unstructured, that you may choose.

This may be an acceptable level of flexibility, given the state of the science in "computer science," but some of these tools have arbitrarily departed from what few standards and conventions do exist. It doesn't make sense for a system to use its own peculiar icon for an included module any more than to allow a CAD/CAM system for electrical engineers to use just any old shape to represent a transistor.

If we use computer-aided software engineering tools, and we call ourselves software engineers, perhaps it's time we acted more like engineers. ■

BIBLIOGRAPHY

Stevens, W. P., G. J. Myers, and L. L. Constantine. "Structured Design." *IBM Systems Journal*, vol. 13, no. 2, 1974.

Yourdon, Edward, and Larry L. Constantine. *Structured Design*. Englewood Cliffs, NJ: Yourdon Press/Prentice-Hall, 1979.

Larry L. Constantine is a software methodologist in Acton, Massachusetts. He is a well-known author and speaker in the field of structured design and analysis. He can be reached on BIX c/o "editors."

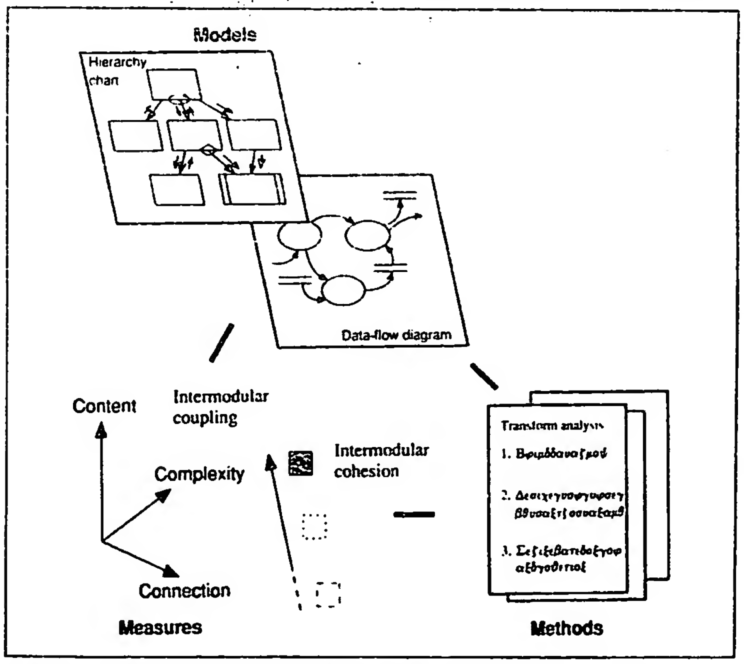


Figure 1: *The troika of structured design: models, methods, and measures.*